CFFI Working Paper No.26-05

One-Shot Traversal for Low-Latency SSD-based Tree Index

Yiheng Tong[1], Minhui Xie[1], Yuanhui Luo[1], Jing Liu[2], Ran Shu[2], Yongqiang Xiong[2], Yunpeng Chai[1]

[1]Renmin University of China, [2]Microsoft Research

Center for Future Financial Innovation
Renmin University of China

One-Shot Traversal for Low-Latency SSD-based Tree Index

Yiheng Tong[1], Minhui Xie[1], Yuanhui Luo[1], Jing Liu[2], Ran Shu[2], Yongqiang Xiong[2], Yunpeng Chai[1]

[1]Renmin University of China, [2]Microsoft Research

Abstract

SSD-based tree indexes (e.g., B+tree) have been widely adoptted in storage and database systems. Driven by modern high-performance SSDs, much prior work has focused on improving their overall throughput. However, when it comes to latency, such approaches still suffer from the inherent bottleneck of pointer chasing during index traversals, so query latency does not benefit directly from faster SSDs. In this paper, we propose Shortcut, a lightweight solution that transforms the conventional wisdom of inherent access dependency into one-shot traversal, thereby achieving low query latency.

Our key idea is to exploit *intra-query parallelism* by training per-level learned indexes as B+tree companion to predict the traversal path in advance, and issuing multiple concurrent I/O requests to prefetch all target nodes in a one-shot manner. The main challenge we deal with is the excessive memory overhead of the additional learned index, which originates not from the machine learning (ML) models, but from two essential structures in it: the key array (for last-mile search) and the value array (termed "mapping array", for mapping model predictions to arbitrary node locations in the file). Shortcut eliminates two arrays through two techniques: *Keyless Learning Method* and *Phantom Mapping Mechanism*, and achieves nearly-zero memory overhead (~0.6%). Evaluations on YCSB and TPC-C show that Shortcut can reduce end-to-end query latency by 26.2% to 64.8%.

1 Introduction

Tree indexes, with B+tree [1–3] as a representative, have long been a cornerstone of data management, powering essential applications such as databases [4, 5], file systems [6, 7], and search engines [8]. A notable strength of B+tree is its efficient on-disk layout, which delivers robust performance on persistent storage: keys are organized in order through a multi-level structure, enabling lookups with only a few I/O operations, while partially filled blocks allow efficient insertions and deletions with less-frequent structural changes.

Meanwhile, driven by advancements in hardware technology (including storage media [9, 10] and interconnects [11, 12]), modern SSD delivers ever-higher throughput [13, 14] (e.g., 2.5M IOPS of Samsung PM1743 PCIe 5.0 SSD [15]). With techniques such as asynchronous I/O and coroutines, modern SSDs can substantially improve the throughput of SSD-based B+tree [13, 16].

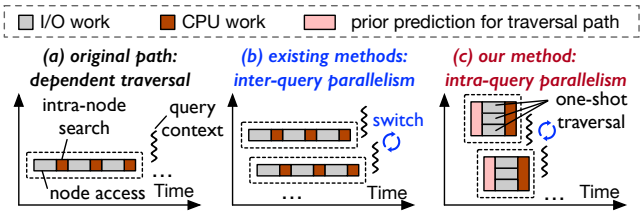


Figure 1: Comparison of different optimization spaces for SSD-based tree index. *(a) The original path follows a dependent traversal. (b) Existing methods (e.g., async I/O, coroutines) exploit inter-query parallelism to improve overall throughput. (c) Our method further exploits intra-query I/O parallelism to reduce per-query latency.*

Unfortunately, the latency of individual indexing operations does not benefit directly from this. The culprit is pointer chasing [17], an inherent issue in traversing hierarchical tree index structures. Such traversals exhibit inherent access dependency (Figure 1a), forcing serial I/O operations and preventing the full utilization of an SSD's parallel processing capabilities. Worse, existing throughput-oriented optimizations [4, 18, 19] primarily rely on *inter-query parallelism* to improve hardware utilization (Figure 1b); they can even degrade latency (e.g., by up to 2.3×) due to additional context-switching overhead.

In this work, we explore the opportunity of *intra-query parallelism*. The key idea is to proactively predict the traversal path before initiating the actual traversal, and then issuing multiple concurrent prefetching I/O requests in a one-shot manner (Figure 1c). As a result, the entire traversal path is predicted by the given search key rather than revealed progressively, enabling query latency reduction proportional to the number of index levels.

Based on machine learning (ML), learned indexes [20–29] offer a promising pathway to this goal. While initially ex-

pected to replace B+tree for their advantage of data awareness, their widespread adoption has been hindered by robustness concern [30]. In this work, we propose a different perspective: treating learned indexes as a companion to B+tree rather than its replacement. Specifically, since index nodes at each level inherently maintain an ordered structure, we can train a per-level learned index to predict node positions with search keys, achieving one-shot traversal.

The implementation of such a companion, however, faces a significant challenge: memory overhead, since maintaining an additional learned index for each tree level prohibitively doubles overall memory usage. Our analysis of a strawman design (named SHORTCUT-BASE) based on a state-of-the-art learned index [23] shows that it is hard to keep the companion completely in memory (§3.3). We identify two inherent space contributors: the key array (49.6%) and the mapping array (49.6%). The key array facilitates searching for accurate positions within an error bound (known as last-mile search [21, 23]). The mapping array translates model predictions to node locations. Both are fundamentally required and cannot be alleviated by simply trading model accuracy or using different learned indexes.

To make SHORTCUT-BASE practical, we propose SHORTCUT, an ultra-space-efficient (i.e., ~0.6% space overhead) B+tree companion, which achieves low query latency by leveraging intra-query I/O parallelism.

SHORTCUT first introduces a *Keyless Learning Method* (§4.2) to train ML models without storing the key array under a *best-effort* paradigm. The key array in traditional learned indexes serves to precisely determine key existence efficiently and do last-mile search. In contrast, in the scenario of SHORTCUT, neither of them is essential, since a prediction failure can fall back to normal traversal path and cause no errors. Therefore, the key array can be safely discarded. Meanwhile, we note that the objective of SHORTCUT is to maximize the number of perfect hits (as opposed to minimizing last-mile search in typical learned indexes). To this end, we tailor a two-phase fitting method that first performs coarse-grained piecewise regression to partition the sorted data to segments, and then refines the slope and intercept within each fitted segment to further retain the prediction accuracy.

Due to the gap between contiguous model predictions and arbitrary node locations (i.e., file offsets), the mapping array cannot be discarded directly like the key array. SHORTCUT introduces a *Phantom Mapping Mechanism* (§4.3) to hide the space overhead of the mapping array with the assistance of file system. Observing that the mapping array's role closely mirrors the file mappings from file offsets to SSD block addresses, we propose to store all nodes at a specific level of the index *contiguously* in a dedicated file. This allows model predictions to be naturally aligned with file offsets; a simple calculation based on the model prediction can determine the file offset. By retrofitting the file mappings in file system, SHORTCUT effectively collapses two layers of address mapping into one layer and makes the mapping array implicit.

Phantom Mapping Mechanism requires contiguous node layouts to maintain a sequential mapping between model predictions and node locations. However, when the tree structure changes (e.g., split/merge), such continuity is disrupted. SHORTCUT employs a *Punching-Aware Update Strategy* (§4.4) to incorporate recent such structural modifications. Specifically, when a node split occurs, the newly created node can be viewed as a "hole" that must be injected into the region adjacent to the split node. This can be handled using hole-punching operations supported by file system. Based on the observation that the hole-punching operations are performed in a blocking manner and incur non-trivial overhead (nearly 1000× slower than `pwrite` as shown in [31]), we propose to decouple them asynchronously from the critical path and defer their execution through batch processing. This effectively amortizes the overhead of hole-punching operations over time. For node merges triggered by delete operations, we can perform the inverse of creating holes by compacting the surrounding data to remove holes.

We implement SHORTCUT as a plug-in and integrate it into LeanStore [4,5,13,32], a widely-used state-of-the-art database storage engine for NVMe SSDs. Our evaluations on YCSB and TPC-C show that SHORTCUT can reduce the end-to-end query latency by 26.2% to 64.8%, while introducing only ~0.6% additional memory consumption.

In summary, this paper makes the following contributions:

- We analyze the latency bottleneck of SSD-based storage systems and the limitation of existing techniques caused by inherent pointer chasing.
- We propose SHORTCUT, a lightweight latency-optimized solution by leveraging ML models, with the goal of achieving intra-query I/O parallelism with high accuracy, space efficiency, and dynamic adaptivity.
- Comprehensive experiments demonstrate the effectiveness of latency reduction and memory efficiency of SHORTCUT.

The rest of this paper is organized as follows. We first cover background on modern SSD-based storage and learned indexes (§2), followed by analysis of the strawman design (§3). We then dive into the SHORTCUT design (§4) and implementation (§5). Finally, we conduct evaluations (§6), discuss related work (§7), and conclude (§8).

2 Background

2.1 Modern SSD-based Storage

The evolution of SSD is primarily driven by the underlying media [9, 10] (such as NAND, Z-NAND, V-NAND, and 3D XPoint) and device interconnects [12] (such as PCIe and CXL). Internally, multiple flash channels are connected to independent flash dies, each containing multiple planes organized into blocks and pages. The basic architecture enables a high degree of internal parallelism [33].

To fully exploit this, most modern storage systems [13, 16] adopt asynchronous I/O interfaces, such as libaio [34],

io_uring [35], and SPDK [36], to increase the number of concurrent I/O requests when interacting with SSDs. These interfaces make it easier to exploit the throughput provided by SSDs. Among them, SPDK achieves the best performance by bypassing the kernel, but suffers from limited ease of use. Since SPDK operates entirely in user space and requires exclusive control over storage devices, it cannot be concurrently shared among multiple independent applications. This limitation restricts its applicability in multi-tenant or general-purpose environments [37, 38]. Consequently, this work primarily focuses on the kernel I/O stack.

From a top-down perspective within the full kernel I/O stack, the I/O execution process involves multiple layers of address translation. The application first generates a file offset, which is translated into a logical block address (LBA) by file system. Once converted to an LBA, the request is sent to the SSD via the NVMe protocol. Finally, the flash translation layer (FTL) translates the LBA into a physical block address (PBA), completing the data access.

2.2 Learned Indexes

Recent studies have introduced learned indexes and explored how to efficiently handle structural modifications under dynamic workloads, with the goal of regarding learned indexes as a promising replacement for traditional indexes [20–29]. The key insight of learned indexes is to view the mapping from keys to their locations as a model that can be fitted using machine learning methods. By doing so, learned indexes can utilize model prediction to accelerate the search process with up to 3.2× lower space overhead compared with traditional indexes (like B+tree) [39, 40].

However, learned indexes still fail to fully replace traditional indexes due to several inherent challenges that are difficult to completely resolve [30]. In particular, learned indexes suffer from poor robustness and significant performance fluctuations on challenging data distributions. In contrast, traditional indexes consistently deliver steady performance regardless of data distributions. In this work, we rethink the role of learned indexes to better leverage their advantages.

3 Motivation and Challenges

3.1 Limitation of Inter-Query Parallelism

To analyze the limitation of existing throughput-oriented optimizations, we use FIO [41] and YCSB [42] to benchmark random read performance on a Samsung PM9A3 SSD by varying levels I/O concurrency. For FIO, we use io_uring as the asynchronous I/O engine and vary the I/O depth. For YCSB, we conduct experiments on LeanStore, which supports coroutine-based query context switching, and vary the number of coroutines assigned to each thread. Both experiments are carried out using 24 threads, and the page size is set to 4 KB.

As shown in Figure 2a, increasing the I/O depth improves throughput but comes at the cost of higher latency. Specifically, for FIO, the P50, P75 and P95 latencies increase by up to 2.3×, 2.2×, and 1.6×, respectively. For YCSB, the per-query latency exhibits a similar trend, which is primarily attributed to I/O accesses during traversals and additional overhead introduced by coroutine scheduling.

Observation #1: Exploiting inter-query parallelism to leverage SSD bandwidth is purely throughput-oriented and can markedly increase end-to-end latency. This is particularly problematic for pointer chasing, where the accumulated I/O latency becomes more sensitive. The underlying reason is that although exiting techniques exploit aggregate bandwidth across multiple queries, they fail to enhance the effective per-query bandwidth utilization due to inherent dependency. Addressing this limitation is the goal of Shortcut.

3.2 Opportunity of Intra-Query Parallelism

In this section, we explore the optimization potential of intra-query parallelism and introduce a strawman solution, named Shortcut-base, to alleviate the latency bottleneck arising from pointer chasing.

To break the conventional wisdom that one complete traversal path can only be revealed progressively as the traversal proceeds level by level, Shortcut-base trains learned indexes as B+tree companion to predict the traversal path in advance. Specifically, for traditional tree-based indexes like B+tree, the nodes at each level can be regarded as a sorted array. Therefore, for each level, by taking the boundary (i.e., upper or lower) keys covered by each node as inputs and the node IDs as outputs, we can construct an efficient learned index to predict the position of the target node at that level. By doing this for each level, the complete traversal path can be predicted before the actual traversal starts.

Figure 3a illustrates three main components of the per-level companion: 1) the ML models are trained to approximate the distribution from keys to node IDs through linear regression. 2) the keys, represented as the key array, are used to perform last-mile search to find the accurate node to access (i.e., node ID), and 3) the associated values, represented as the mapping array, store the mapping from node IDs to node locations (i.e., file offsets), which are subsequently translated by file system and delivered to SSDs as prefetching I/O requests. In fact, the key array and mapping array are the key-value pairs in the leaf nodes of learned indexes.

We implement Shortcut-base based on the state-of-the-art updatable learned index ALEX [23] and integrate it into LeanStore. Figure 2b presents the query latency CDF and achieved throughput under 24 threads and 4 coroutines. Compared with LeanStore, Shortcut-base reduces the P50 and P95 latencies by 17.9% and 20.7%, respectively. It also improves overall throughput by 23.4%.

Observation #2:* Shortcut-base *can achieve a notable reduction in per-query latency without affecting throughput. This is achieved by training learned indexes as B+tree companion to predict the traversal path in advance, enabling parallelized I/O requests within a single query. The reduced la-

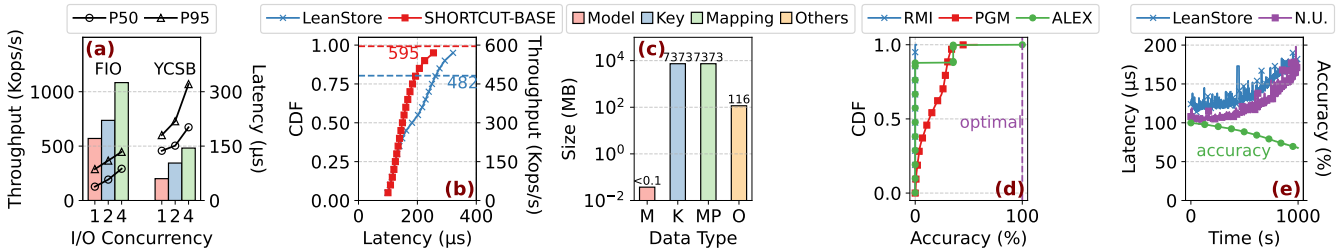


Figure 2: Motivation. *(a) Trends in throughput and latency by varying I/O concurrency on FIO (iodepth) and YCSB (number of coroutines in LeanStore). (b) Performance comparison of LeanStore and* Shortcut-base*. (c) Space consumption breakdown of* Shortcut-base*. (d) Prediction accuracy CDF of existing training methods by eliminating the key array directly and relying on the predicted position. (e) Latency and prefetching accuracy changes under insert-heavy workload (50% read, 50% insert) with latest distribution. N.U. refers to* Shortcut-base *not being updated to reflect the structural changes of the original index.*

tency further improves throughput by allowing shorter critical sections for better concurrency.

3.3 Challenges

While our analysis demonstrates the potential performance benefits of Shortcut-base, we also draw two lessons showing that it is not yet practical.

Lesson #1:* Shortcut-base *suffers from excessive memory overhead due to the additional learned indexes. We observe that the memory consumption of Shortcut-base is less efficient than expected, as described in §2.2. Figure 2c presents a detailed breakdown. The key array and the mapping array account for the majority of the total memory footprint, consuming 49.6% and 49.6%, respectively, while the ML models and other metadata contribute only 0.8%. Such excessive memory consumption is difficult to warrant in real-world scenarios. For example, mainstream storage-optimized cloud instances (Amazon EC2) exhibit a memory-to-SSD capacity ratio of roughly 1% [43]. In our test, the total B+tree size is 2.6 TB, and the inner nodes consume 26.3 GB (~1%). Further, a substantial portion of memory is allocated for data cache and other runtime components [37], leaving less available memory for index cache. Thus, Shortcut-base competes heavily with index cache.

We then take a deep dive into the root cause of the space inefficiency in Shortcut-base. Existing learned indexes are designed as replacements for B+tree. As such, all of them fundamentally require the key array for correctness (last-mile search) and mapping array to store associated values (i.e., arbitrary node locations in Shortcut-base). Both cannot be alleviated by simply trading model accuracy or using different learned indexes.

For the key array, it is possible to discard it directly and only rely on the predicted position by ML models, but this may lead to a decrease in prediction accuracy. To quantify this effect, we evaluate three training methods commonly used in learned indexes [20,21,23]. As shown in Figure 2d, the overall highest accuracy is achieved by PGM [21]. Nevertheless, 99% of ML models exhibit an accuracy below 36%. The severe accuracy degradation motivates the need for a tailored training method for our scenario.

For the mapping array, it cannot be discarded directly as the functionality for address translation must be preserved. Otherwise, an I/O request would be unable to determine the address for data access. Therefore, achieving space efficiency for the mapping array also remains a challenge.

Lesson #2:* Shortcut-base *suffers from limited performance benefits if it fails to adapt to structural modifications in the original index. As Figure 2e shows, when new keys are inserted, the latency reduction of Shortcut-base gradually diminishes if it is not updated to reflect the structural changes (depicted as N.U.) in the original index caused by node splits. The underlying reason is that the prefetching accuracy during runtime gradually degrades over time as node splits occur. This motivates the need for an adaptive design capable of coping with structural modifications under dynamic workloads.

Moreover, Figure 2e provides an important insight that updating Shortcut-base immediately after a node split is unnecessary, as the correctness of index queries does not rely on prefetching. Within a certain tolerance for inaccuracy, Shortcut-base can still deliver notable performance benefits.

Building on the lessons learned from Shortcut-base, we summarize three main challenges that must be addressed:

- **High accuracy**. The prediction accuracy for the traversal path is essential to achieve the benefits of parallel prefetching I/O requests. However, simply dropping the key array would be disastrous for accuracy.
- **Space efficiency**. Given the limited memory budget, allocating additional space to predict the traversal path becomes prohibitively expensive and impractical if it entails a large memory footprint.
- **Dynamic adaptivity**. Since the original index may undergo structural modifications under dynamic workloads, these changes should be accurately tracked.

4 Design

4.1 Architecture

Motivated by our analysis, we propose the design of Shortcut, which maintains **an ultra-space-efficient** approximate

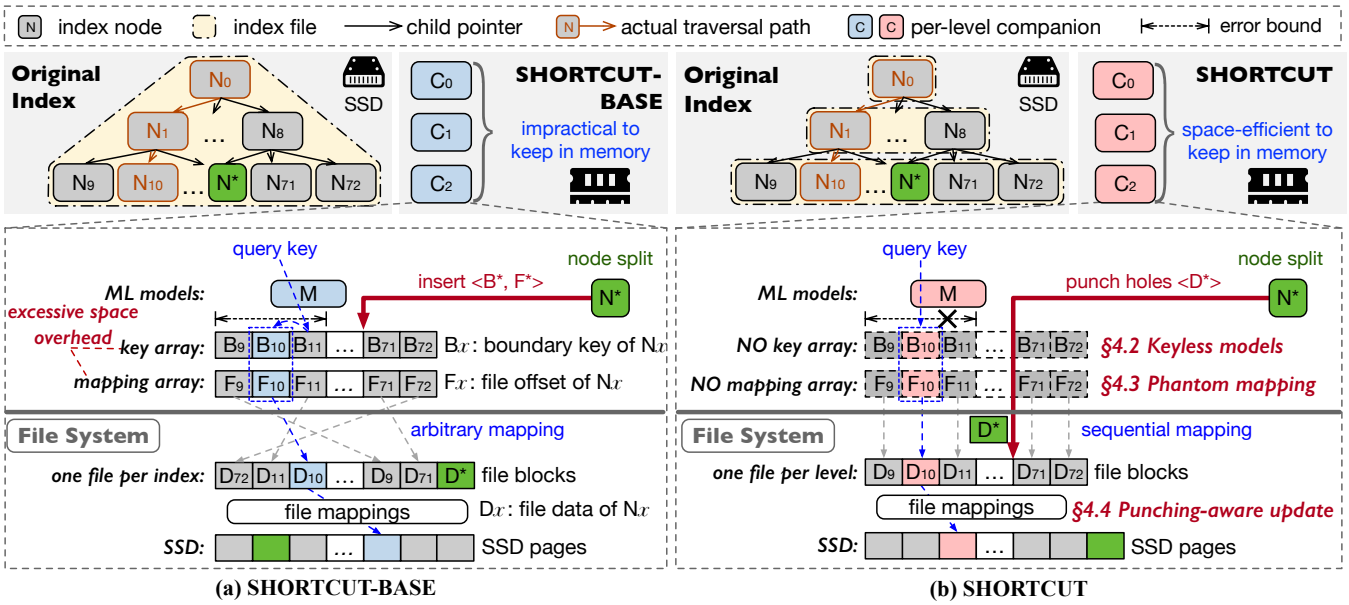


Figure 3: Overview architecture of two solutions to achieve one-shot traversal. *(a)* SHORTCUT-BASE*: The strawman solution suffers from excessive space overhead. (b)* SHORTCUT*: Make the strawman into practicability by three techniques .*

mapping from search keys to their traversal paths. Our key insight for minimizing memory consumption is to tailor ML models to capture only necessary information for one-shot traversal, and to leverage internal file system mechanisms to map keys to node locations instead of storing this information in SHORTCUT.

The architecture of SHORTCUT is shown in Figure 3b. To address the aforementioned key challenges, SHORTCUT introduces three techniques.

1) *Keyless models.* SHORTCUT prefetches all target nodes based on the predictions of keyless ML models. We propose a learning method (§4.2) that strives for maximal, yet achievable, prediction accuracy under a best-effort paradigm while eliminating the need to store the key array.

2) *Phantom mapping.* SHORTCUT retrofits the file mappings in file system (§4.3) to hide the space overhead of the mapping array. By storing index nodes contiguously in the file, the arbitrary mapping from model predictions to file offsets is converted into a sequential mapping, thus making the mapping array implicit.

3) *Punching-aware update.* SHORTCUT employs an efficient strategy (§4.4) to preserve the contiguous node layout, which is disrupted by structural modifications in the original index. By optimizing punching operations supported by file system, SHORTCUT can adapt to these changes asynchronously.

Overall, the basic workflow operates as follows: 1) Before initiating the traversal of the original index for a given key, SHORTCUT first predicts all target nodes, which are subsequently issued as asynchronous prefetching I/O requests to SSDs. 2) During the actual traversal from the root, if a target node is not yet cached, the system checks for a prior prefetching request. On a hit, it switches to another query and subsequently polls for completion. Otherwise, it falls back to the regular path and immediately issues a standard I/O request to access the node. 3) When a node split occurs, the file system will allocate available space for the newly created node (e.g., at the end of the file). The event is temporarily stashed and later reflected in SHORTCUT.

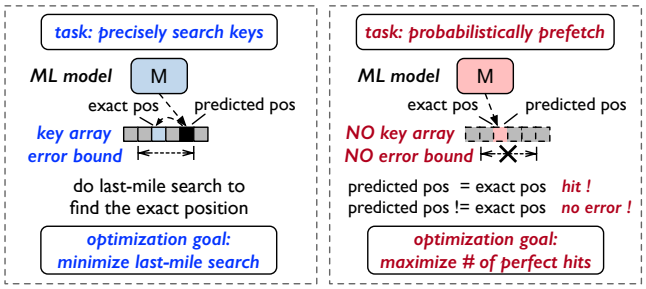
4.2 Keyless Learning Method

We revisit the differences between the ML models used in prior learned indexes and those used in our one-shot traversal scenario. The key array serves two purposes in learned indexes: (1) for performing last-mile search to locate the corresponding key–value pair precisely, and (2) for determining with certainty whether a queried key exists.

However, in one-shot traversal, neither of these functionalities is essential: (1) a prediction failure does not cause any errors, and (2) regardless of whether the key exists, the same path will be accessed by the subsequent actual traversal. Thus, storing the key array becomes unnecessary.

We propose a *keyless learning method* – its key idea is to achieve high prediction accuracy without storing keys, following a *best-effort paradigm*. Figure 4 illustrates a keyless model in SHORTCUT and compares it with a typical model used in learned indexes. They differ in three aspects: (1) optimization goal, (2) training data collection, and (3) fitting method. We describe each below.

Optimization goal. Given a key set $\mathcal{K}$, the goal of a typical model in learned indexes is to maximize the search performance (i.e., overall throughput), which corresponds to minimizing the distance of last-mile search to reduce the number



(a) typical model in learned indexes **(b) keyless model in SHORTCUT**

Figure 4: Comparison of ML models under different scenarios. *(a) The typical model in learned indexes must ensure correctness for searching keys. (b) The keyless model in* SHORTCUT *is expected to hit and incurs no error when misses.*

of key comparisons. This can be formalized as:

$$\text{minimize last-mile search} = \min_{M} \sum_{x \in \mathcal{K}} |M(x) - y| \qquad (1)$$

where $M(x)$ denotes the position predicted by model M for a given key x, and y denotes the true position.

In contrast, in one-shot traversal, inaccurate predictions, no matter how closely the predicted position approximates the actual position, result in prefetching misses. Thus, the goal of SHORTCUT is to maximize the number of perfect hits, which can be formalized as:

$$\text{maximize \# of perfect hits} = \max_{M} \sum_{x \in \mathcal{K}} I(M(x) = y) \qquad (2)$$

where $I()$ denotes an indicator variable, which equals 1 if the condition is true and 0 otherwise.

Collecting the training data. To train keyless models from scratch, SHORTCUT needs to collect the boundary keys for each node as the training data. The boundary key can be chosen as either the upper or lower key covered by the subtree rooted at the current node. Note that the maximal or minimal key within a node is not sufficient to represent its boundary, because the actual boundary lies beyond this range, which is determined by its parent or upper-level ancestors, as shown in Figure 5. Thus, the training data collection can be efficiently implemented by traversing all inner nodes, without the need for a time-consuming scan of all leaf nodes. Moreover, selecting only one key per node can also simplify the fitting difficulty and reduce the impact of local outliers.

Fitting method. After collecting the training data, we have obtained the per-level sorted key arrays. We train ML models for each level through a two-phase fitting method. In the first phase (coarse-grained fitting), similar to a learned index [21, 44], the per-level dataset is further partitioned into multiple segments (i.e., multiple linear pieces), via piecewise linear regression. For each point <*key, node ID*>, we determine whether it can be incorporated into the current segment by adjusting the slope and intercept. This phase yields the

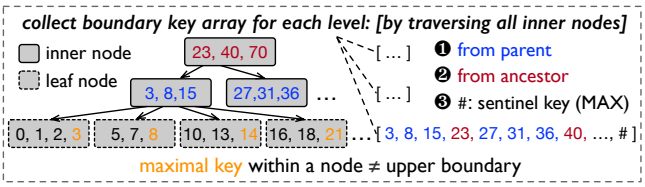


Figure 5: Collecting the training data. *Taking the upper boundary key array at the leaf level as an example, 3, 8, 15, 27, 31, and 36 can be collected from their parent nodes, while 23, 40, and 70 are from their ancestors (the root node).*

minimal number of segments required to approximate the distribution based on a predefined error bound, which corresponds to the lowest space consumption for models.

In the second phase (fine-grained fitting), we propose to fine tune the fitted curve for each segment, aiming to maximize the object function Equations 2. However, it is infeasible to directly optimize this objective, as $I(M(x) = y)$ is discrete, discontinuous, and non-differentiable, thereby precluding the use of conventional gradient descent or least squares approaches. In this work, we replace the indicator function with the squared error and then perform optimization via least squares minimization.

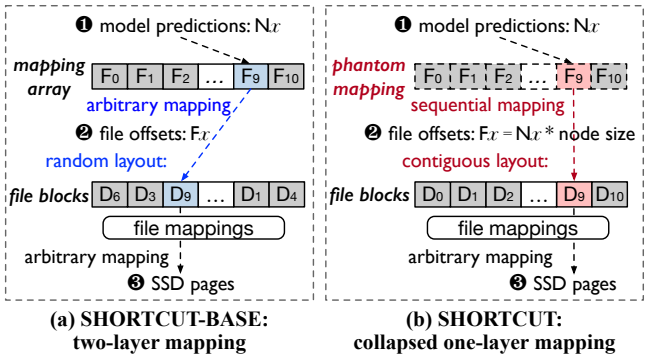
Our experiments (§6.3) demonstrate that, the keyless models trained via the two-phase fitting method achieve around 50% average prediction accuracy, which is sufficient for SHORTCUT.

4.3 Phantom Mapping Mechanism

While the key array can be discarded, the mapping array is indispensable as it bridges the gap between the contiguous, logical predictions of ML models, and the random, physical file offsets of tree nodes. This stems from the fact that the model predictions are typically monotonically increasing, whereas the file offsets are often allocated in a fragmented and non-contiguous manner. As discussed in §3.3, the space overhead of an explicit mapping array is substantial, which is on the same order of magnitude as the number of leaf nodes in the original index. To address this, SHORTCUT proposes a *phantom mapping mechanism* to hide the space overhead of such a mapping array, as Figure 6 shows.

Contiguous node layout is the key. Based on the observation that the mappping array serves to map model predictions arbitrarily to file offsets, SHORTCUT proposes to store nodes at each level contiguously within the file, converting the arbitrary mapping into a sequential mapping. The contiguous node layout implicitly serves the role of a mapping array, as the file offset of a node can be directly computed by multiplying its node ID by the node size (e.g., 4 KB). This effectively collapses two layers of address translation, from node IDs to file offsets and from file offsets to LBAs, into a single step, thereby avoiding the space overhead of maintaining an explicit mapping array.

Hole-punching operations can preserve the required



(a) SHORTCUT-BASE: two-layer mapping **(b) SHORTCUT: collapsed one-layer mapping**

Figure 6: Comparison of address translation process from model predictions to SSD pages. *(a) The two-layer address translation includes two arbitrary mappings (mapping array and file mappings). (b) The phantom mapping converts the arbitrary mapping from model predictions to file offsets into a sequential mapping by storing nodes contiguously in the file.*

contiguity. The contiguous node layout can be disrupted by structural modifications in the original index such as node splits and merges, which can be preserved through hole-punching operations. Linux file systems such as Ext4 and XFS support hole punching, which allow a hole of a specified length to be created (i.e., insert-range) or removed (i.e., collapse-range) at a given file offset. All subsequent file offsets are shifted by the hole length, without the need to move actual data. Only the metadata (i.e., file mappings) is updated, which is typically stored in an efficient index in file system.

Separation of levels into distinct files. To minimize interference across different index levels, all nodes within the same level are stored separately in a dedicated file. This ensures that structural modifications occurring within one level induce no cascading disruptions at other levels.

4.4 Punching-Aware Update Strategy

The required contiguity in phantom mapping can be preserved through hole-punching operations in file system. Unfortunately, these operations complicate updates, as they are performed in a blocking manner and suffer from poor performance. To address this, we propose a *punching-aware update strategy*, which decouples hole punching from the critical path of foreground queries in an asynchronous manner.

Updates do not interfere with foreground query execution. When a node split/merge occurs, we do not immediately create or remove a hole at the corresponding position. Instead, we log the event in a customized stash queue and defer the actual execution of multiple hole-punching operations in batches, allowing the cost to be amortized. During asynchronous batch processing, SHORTCUT is temporarily disabled, while foreground queries can continue to proceed along the regular traversal path without any interruption.

Note that the file offset of node position is affected by hole-punching operations. Therefore, in the original index, we use LBAs rather than file offsets as child pointers. This only requires the file system to expose a `GET_LBA` interface [31] for the application to get the LBA given a file offset, without introducing any side effects.

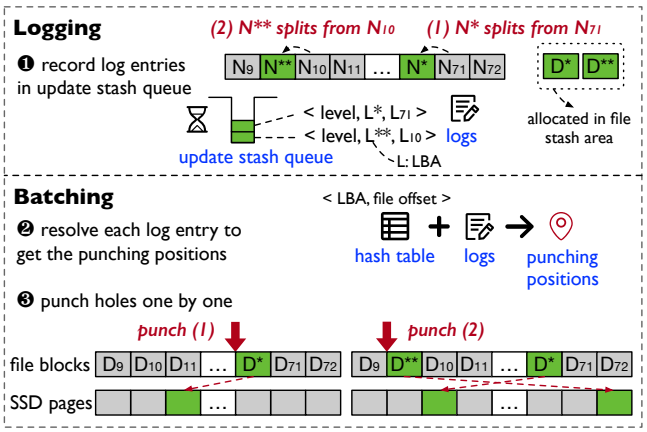


Figure 7: Design of punching-aware update strategy.

The detailed logging and batching procedures are shown in Figure 7 and described as follows:

- **Logging.** Upon a structural change to a node, we record a log entry in the stash queue, containing the current level, the LBAs of the original node and the newly created or removed node. The accumulated log entries serve as metadata to guide subsequent hole-punching operations.
- **Batching.** Once the size of the stash queue reaches a predefined threshold (5% new split nodes), the corresponding batch of hole-punching operations is executed asynchronously in the background. Based on file mappings, we construct a temporary hash table with LBAs as keys and file offsets as values. This allows each log entry to be efficiently resolved to the exact file offsets where holes need to be created or removed. According to the punching positions, we punch each hole one by one. Finally, to update the ML models, we can efficiently collect the latest key array by traversing only the inner nodes, as described in §4.2. The models for each level are then retrained accordingly.

Optimization: selective prefetching. Although our proposed update strategy enables SHORTCUT to efficiently track structural changes in the original index, the prediction accuracy may still exhibit sharp drops in certain models, particularly under irregular or highly dynamic key distributions. To address this, SHORTCUT incorporates a feedback mechanism at the end of each traversal to track run-time prefetching accuracy. Based on the observed statistics, we can selectively invalidate models with low accuracy, allowing for subsequent background adjustments.

4.5 Concurrency

The concurrency issue of SHORTCUT arises in two cases. First, the dedicated background thread for collecting training data does not acquire any latches on the B+tree (similar to normal lookups) and checks versions to detect read-write conflict.

Additionally, since nodes previously traversed may undergo new structural changes, we record them as temporary logs. This ensures that, once the collection is completed, the state of the collected key array (along with the temporary logs) and the update stash queue remain consistent.

Second, during the batching of hole-punching operations, concurrent structural changes may occur; we record these changes in a new stash queue, to be applied when SHORTCUT is next triggered for an update.

4.6 Recovery

SHORTCUT is lightweight enough to be rebuilt on demand with minimal time and space overhead. Thus, it can be treated as a non-persistent structure. If durability is required, we can choose to persist the ML models during I/O idle periods. The persistence of hole-punching operations, which modify file mappings, is fully managed by the underlying file system. The file system's logging mechanism ensures recovery to a consistent state after a crash.

5 Implementation

We implement SHORTCUT as a plug-in that can be seamlessly integrated into a real system with minimal code intrusion. The two-phase fitting method for training keyless models is implemented based on [21], which can provide provably space bounds in the worst case. We build on the underlying file system used in prior work [31], which provides the `GET_LBA` interface. It also supports high-throughput hole-punching operations, which facilitates our asynchronous punching and batching techniques.

Integration. LeanStore [4, 5, 13, 32] is a widely-used state-of-the-art database storage engine for NVMe SSDs, which provides over 2× throughput than other systems such as WiredTiger [45]. The branch that we forked [13] also supports cooperative multitasking based on Boost coroutine library [46]. Our modifications to LeanStore comprise around 500 LoC, which primarily involve inserting barriers to redirect the regular traversal path to one-shot path, managing the allocation and reclamation of memory resources for prefetching, and coordinating interactions with the memory cache.

Caching. To interact with the memory cache of index nodes, we use a hash table to exclude those cached nodes from prefetching to reduce wasteful usage of SSD bandwidth. Meanwhile, the prefetched nodes are initially placed into a temporary prefetch pool in memory. Only valid (i.e, accurate) prefetched nodes are subsequently admitted into the memory cache and become visible to cache accesses, which can also prevent cache pollution caused by inaccurate prefetching.

6 Evaluation

6.1 Experimental Setup

Environment. We conduct all experiments on a dual-socket machine equipped with two 28-core Intel Xeon CPU Gold 6330 Processors, 500 GB of DRAM, and a 3.84 TB Samsung PM9A3 SSD. The code is compiled using GCC 11.4.0 with the optimization of `-O3` and run on Ubuntu 22.04.5 with Linux kernel version 5.15.0-136. Hyper-threading is disabled, and all threads are bound to a single NUMA node.

Workloads. We use the Yahoo! Cloud Serving Benchmark (YCSB) [42] as our microbenchmark, which contains six workloads. Both uniform and zipfian ($\alpha = 0.99$) distributions are evaluated, which are generated in an open-loop manner with request arrivals following a Poisson distribution. For each workload, we initialize the B+tree with a target size of 100 GB (about 3.3 billion 8B-8B key-value pairs), resulting in a tree height of 5. The size of B+tree node is set to 4 KB and the node fanout is 198. Unless otherwise stated, we allocate the memory cache sized at 1% [37, 43] of the B+tree and warm it up before running each test.

Datasets. In addition to the synthetic (i.e., linear) dataset used in YCSB, we also use four real-world datasets with increasing fitting difficulty [39]: 1) *covid*: Tweet IDs with COVID-19 [47], 2) *libio*: repository IDs from Libraries.io [48], 3) *genome*: loci pairs in human chromosomes [49], and 4) *osm*: sampled OpenStreetMap locations [50]. Each dataset contains 200M 8B unsigned integer keys.

Comparison Systems. We compare SHORTCUT with the following baselines: 1) LeanStore: the default baseline that performs dependent traversals on B+tree. 2) SHORTCUT-BASE: the strawman solution that achieves perfectly accurate prefetching but incurs excessive memory overhead. 3) SHORTCUT-SYNC: a variant of SHORTCUT that performs synchronous hole-punching operations to preserve the required contiguity in the phantom mapping mechanism. All of them use io_uring [35] as the asynchronous I/O interface.

We also compare with three existing systems optimized with learned indexes: DLI [44], Bourbon [51] and XStore [52]. DLI is an efficient on-disk learned index to fully leverage the characteristics of disk storage. Bourbon utilizes learned indexes for LSM-trees to provide fast lookups. XStore is a learned cache for RDMA-based tree index by training ML models over leaf nodes[1]. The three competitors use background threads for model (re)training; we keep their thread counts consistent for fair comparison.

All systems set `O_DIRECT` to bypass the page cache in the kernel. Unless stated otherwise, we run 24 worker threads for all experiments, with dedicated threads for workload generation and background profiling. Except when comparing with three exsiting systems, each worker thread use 4 coroutines.

6.2 YCSB Performance

In this section, we evaluate the end-to-end query latency and overall throughput of SHORTCUT, LeanStore and SHORTCUT-BASE using YCSB. Notably, SHORTCUT-BASE represents the optimal

[1] Although XStore is not originally proposed for SSD scenario, we implement its core idea under static workloads: fetching all target leaf nodes within the error bound of model predictions to ensure searching correctness, thereby replacing the traversal of the whole tree structure.

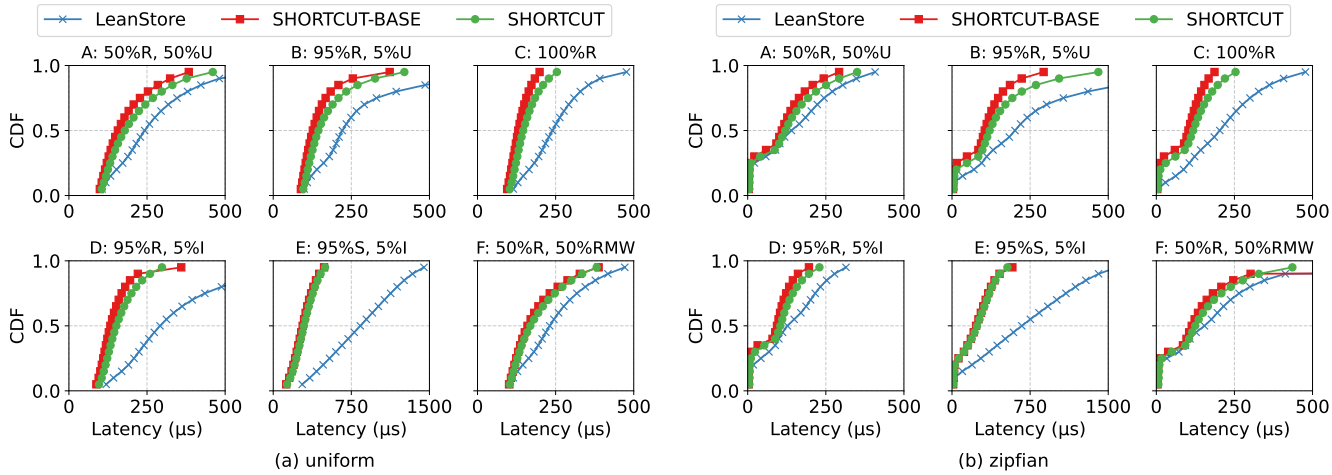


Figure 8: (Exp #1) End-to-end query latency CDF under YCSB workloads. *R: Read, U: Update, I: Insert, S: Scan, RMW: Read-modify-write. (a) uniform, (b) zipfian.*

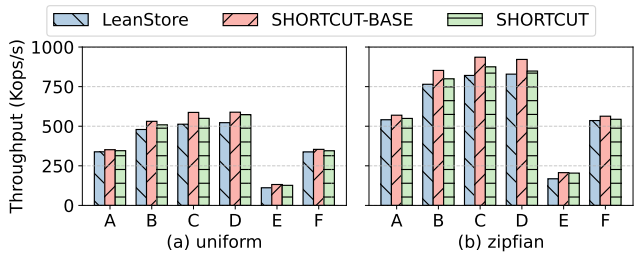


Figure 9: (Exp #2) Throughput under YCSB workloads. *(a) uniform, (b) zipfian.*

performance improvement of SHORTCUT at the cost of excessive space overhead.

Exp #1: End-to-end query latency. Figure 8 shows the end-to-end query latency CDF under YCSB workloads. We can make the following observations: 1) Compared with LeanStore, SHORTCUT consistently achieves lower latencies, with reductions of 26.2% to 64.8% and 19.1% to 69.9% at the P50 and P95 percentiles, respectively. This benefits from converting the sequential I/O within individual queries into parallel prefetching requests based on the predictions of traversal path. 2) Compared with SHORTCUT-BASE, SHORTCUT exhibits a modest latency gap from 1.8% to 21.8%. This is because SHORTCUT-BASE retains the key array to enable perfectly accurate prefetching, whereas SHORTCUT relies on keyless models, inevitably introducing some prefetching misses. 3) Compared with the uniform distribution, the latency CDF curve under the zipfian distribution shifts upward. This is because a portion of hot nodes can be served directly from the memory cache, avoiding to trigger I/O operations.

Exp #2: Overall throughput. Figure 9 shows the throughput under YCSB workloads. Similar to latency results, the throughput of SHORTCUT lies between that of LeanStore and SHORTCUT-BASE. Under uniform and zipfian distributions, SHORTCUT outperforms LeanStore with slight throughput improvements of 2.1% to 13.9% and 1.5% to 21.4%, respectively. This is because SHORTCUT suffers few misses on YCSB synthetic dataset. The achieved lower latency enables shorter critical sections, allowing better concurrency and ultimately leading to a marginal throughput increase.

Exp #3: Insert-heavy workload. To demonstrate the effective update of SHORTCUT under dynamic workloads, Figure 10 shows the performance timeline of insert-heavy workload (50% read, 50% insert) with latest distribution after loading 300M key-value pairs. As time elapses, the prediction accuracy of SHORTCUT remains stable around 58% to 62% through periodic updates asynchronously, thereby sustaining throughput and latency without decline. The training of SHORTCUT takes 1s with only one background thread, and retraining is triggered every ~100s. In contrast, SHORTCUT-SYNC suffers from a throughput and latency drop of over 1000× due to its blocking execution of hole-punching operations, which severely stalls foreground queries.

6.3 Real-world Datasets

In this section, we evaluate the effectiveness of SHORTCUT on real-world datasets with various fitting difficulties.

Exp #4: Throughput and latency. Figure 11 shows the throughput, P50 latency, and P95 latency on four real-world datasets. The linear dataset acts as a reference, and the workload is generated by YCSB. Compared with LeanStore, SHORTCUT retains similar throughput, although with 10% bandwidth waste caused by mispredictions. For P50 latency, SHORTCUT achieves 15.4% to 66.7%, 17.3% to 63.9%, 15.7% to 61.1%, and 14.2% to 68.9% latency reduction on *covid*, *libio*, *genome*, and *osm*, respectively. This is attributed to the stable prediction accuracy under various data distributions, which will be analyzed later. The P95 latency shows less noticeable improvement due to the combined effect of prefetching misses and hardware factors (e.g., SSD garbage collection).

Exp #5: Best-effort prediction accuracy. To provide a

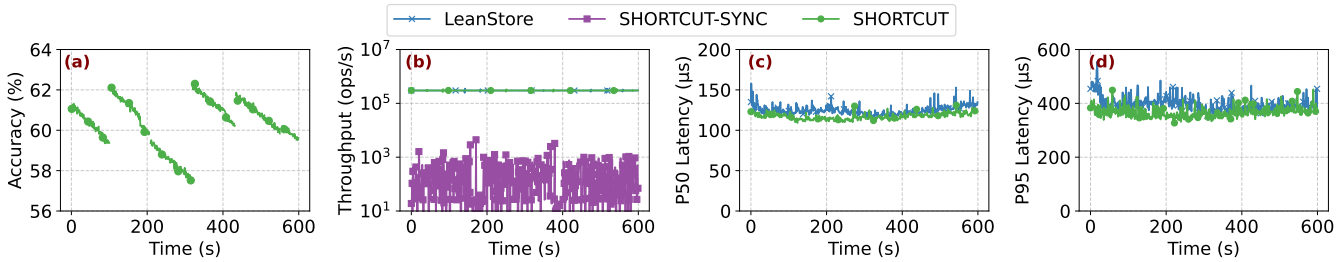


Figure 10: (Exp #3) The performance timeline under insert-heavy workload (50% read, 50% insert) with latest distribution. *(a) Prefetching accuracy during runtime, (b) Throughput, (c) P50 Latency, (d) P95 Latency. The latency of* SHORTCUT-SYNC *is omitted due to its sharp increase.*

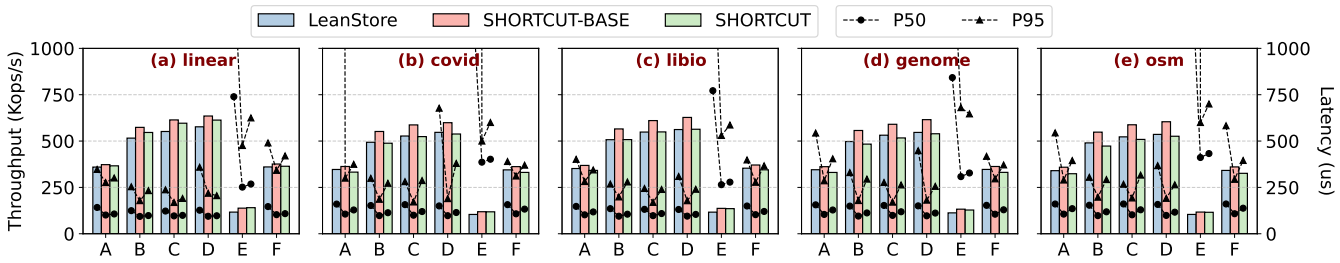


Figure 11: (Exp #4) Performance on real-world datasets. *OSM is recognized as the hardest dataset [39].*

fine-grained view of the prediction accuracy achieved by the ML models in SHORTCUT, we scan all loaded keys and measure the prediction accuracy for each fitted segment (i.e., each model). As shown in Figure 12, the prediction accuracy exhibits distinct patterns across various data distributions. For the linear dataset, all of the segments achieve over 50% prediction accuracy. For harder distributions on real-world datasets, although the prediction accuracy exhibits a slight downward trend as the fitting difficulty increases, the average accuracy remains stable around 50%. This is attributed to the refinement of the slope and intercept for each fitted segment in our tailored training method.

6.4 Comparison with existing systems

In this section, we compare SHORTCUT with DLI and Bourbon to evaluate the performance of different SSD-based storage systems optimized with learned indexes. We also compare SHORTCUT with the core idea of XStore on SSD to evaluate the performance of different strategies to fetch leaf nodes.

Exp #6: Comparison with DLI, Bourbon and XStore. To ensure a fair comparison for all competitors, we configure approximately the same memory budget of 80 MB (1% of the B+tree size in SHORTCUT), used as the memory cache for index nodes in SHORTCUT and XStore, the dynamic stage for accommodating insertions in DLI, and the memtable in Bourbon, respectively. For SHORTCUT, each worker thread uses 1 coroutine, as coroutines are not supported by the other competitors. We first load 200M key-value pairs, and then run two YCSB workloads with different read/write ratios. As shown in Figure 13, we make the following observations:

1) For workload A (100% read), SHORTCUT achieves 1.4× higher throughput than Bourbon and comparable throughput to DLI under uniform distribution. When the distribution shifts to zipfian, Bourbon and DLI maintain performance similar to that under uniform distribution, while SHORTCUT achieves 1.4× higher throughput. This is because SHORTCUT can utilize memory cache more effectively for lookups, whereas Bourbon and DLI mainly use memory as a write buffer and are unable to cache read-intensive hotspots.

2) For workload D (95% read, 5% insert), the P95 tail latency of DLI fluctuates significantly under uniform distribution. The reason is that although the dynamic stage in memory can accommodates new insertions, the triggered merging procedure into the static stage on SSD adversely impacts performance. The throughput of DLI even drops by more than 10× under latest distribution. The throughput and latency of Bourbon are comparable to those of SHORTCUT, with a portion of the P95 latency fluctuations arising from background compactions in LSM-trees.

3) Compared with XStore under workload A (100% read), SHORTCUT achieves 1.7× and 2.4× higher throughput under uniform and zipfian distribution, respectively. We explain this from two aspects. First, SHORTCUT prefetches only one node with high accuracy for each level of the traversal path, and excludes already cached nodes to further avoid wasting SSD bandwidth, whereas XStore prefetches multiple leaf nodes based on the error bound of ML models. Our profiling result shows that XStore consumes 1.9× more SSD bandwidth than

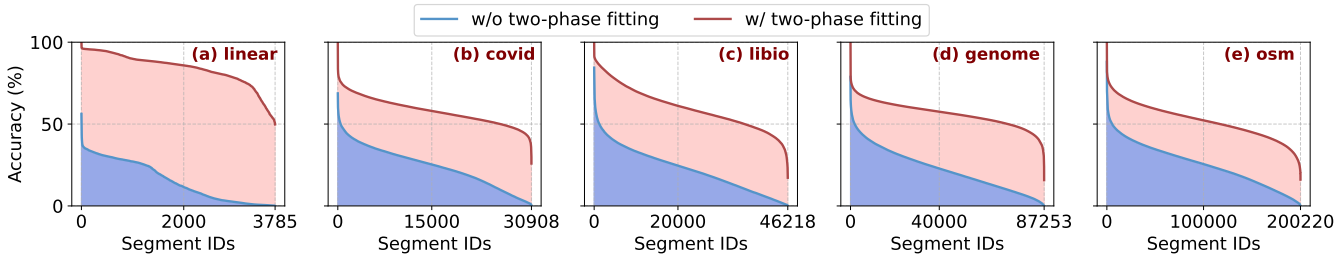


Figure 12: (Exp #5) Per-segment prediction accuracy under various data distributions.

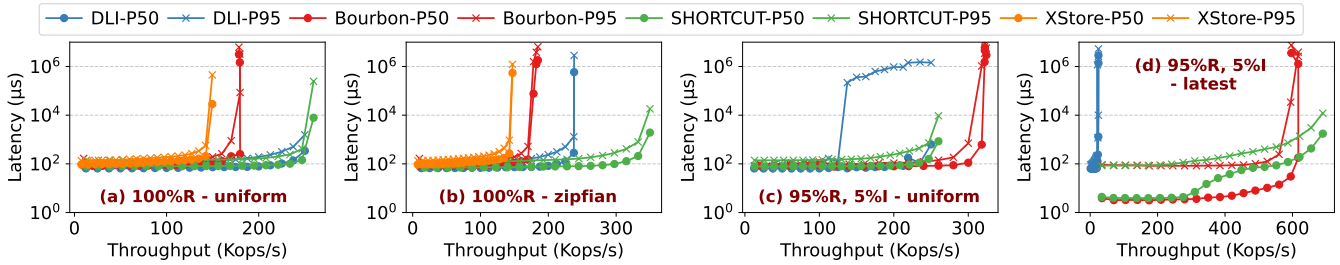


Figure 13: (Exp #6) Performance comparison with existing systems. *All competitors are configured with approximately the same memory budget to ensure a fair comparison. The throughput and latency are measured by gradually increasing the open-loop arrival rate in each test (i.e., progressively applying higher stress).*

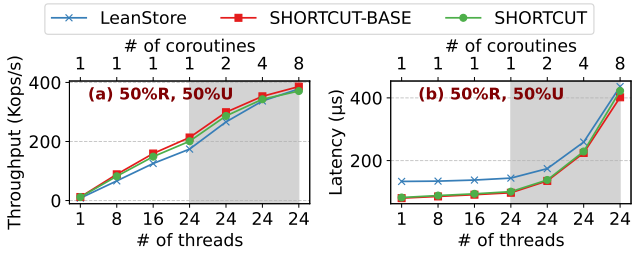


Figure 14: (Exp #7) Sensitivity: Impact of concurrency. *Workoad: A - uniform. (a) Throughput, (b) P50 latency.*

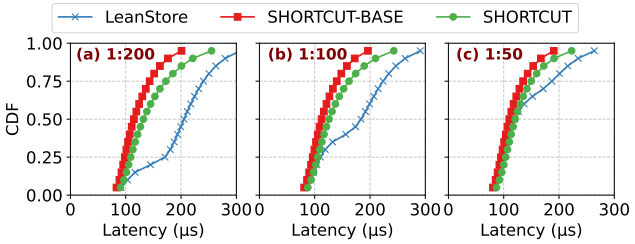


Figure 15: (Exp #8) Sensitivity: Impact of memory cache size. *Workload: C - uniform. Cache ratio: (a) 1:200, (b) 1:100, (c) 1:50*

Shortcut, leading to poor performance under uniform distribution. Second, XStore is also unable to effectively interact with the memory cache for index nodes, since the entire dependent traversal is bypassed. For a given search key, it cannot determine which leaf node to access, and check whether it has already been cached. As a result, XStore delivers similar performance under both uniform and zipfian distributions.

6.5 Sensitivity Analysis

In this section, we vary different experiment parameters to evaluate the sensitivity of Shortcut to these changes.

Exp #7: Concurrency. Figure 14 shows the impact of concurrency by varying the number of threads and coroutines. When the number of coroutines is fixed at 1 and the number of threads increases, throughput improves while latency shows only a slight increase. When the number of threads is fixed at 24 and the number of coroutines increases, throughput continues to improve, but latency increases significantly. Using 24 threads with 4 coroutines represents a sweet point, as further increasing the number of coroutines yields only a marginal throughput improvement of 9.3% but causing 2× latency.

Exp #8: Memory cache size. Figure 15 shows the impact of memory cache size on the performance benefits of Shortcut. When the memory cache size is enlarged, the latency reduction benefits shift toward higher percentiles, which primarily result from multiple I/O requests within a query that can be optimized by Shortcut. When the memory size is sufficient to cache all inner nodes, it falls outside the optimization scope of Shortcut. In such scenarios, one-shot traversal can be disabled and will not cause any performance degradation.

Exp #9: Model size. Figure 16 shows the variation of prediction accuracy and throughput performance on linear dataset with different model size by tuning the error bound parameter ε during the first fitting phase. As ε increases, the number of fitted segments decreases, resulting in a smaller model size. However, the prediction accuracy is also sensitive

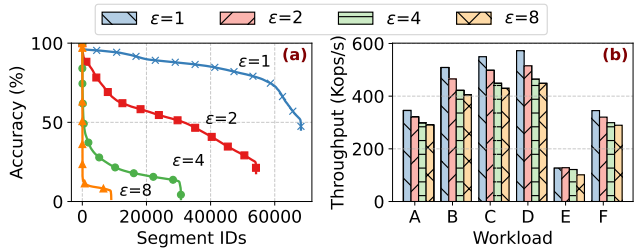


Figure 16: (Exp #9) Sensitivity: Impact of model size. *(a) Per-segment prediction accuracy, (b) Throughput. The model size is implicitly reflected as segment IDs.*

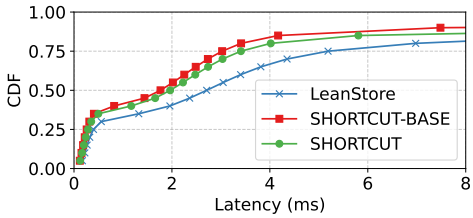


Figure 17: (Exp #10) TPC-C Performance.

to ε, which leads to a throughput decline of up to 21.9% when ε increases to 8. We argue that even with $\varepsilon = 1$ (the most accurate case), the additional space consumption of the models is negligible 1 MB, where the models can most precisely detect local outliers and partition them into distinct segments for further fine tuning within each segment.

6.6 TPC-C Performance

In this section, we demonstrate the effectiveness of Shortcut under production workload TPC-C [53] and analyze its memory efficiency.

Exp #10: Transaction latency. We deploy 2000 warehouses (320 GB in total) and build indexes on the primary keys of each table. The memory cache size is configured to 1.6 GB. Each transaction requires multiple index queries. As shown in Figure 17, compared with LeanStore, Shortcut reduces the P50 and P95 latency by 27.5% and 10.3%, respectively.

Exp #11: Memory efficiency. Table 1 further shows the comparison of memory efficiency in the aforementioned experiment using QPS/GB, with the throughput evaluated under a predefined SLA. We define the SLA as requiring latency below 5 ms. Compared to LeanStore, Shortcut-base achieves 1.19× higher QPS but incurs 1.73× higher memory overhead. In contrast, Shortcut achieves 1.13× higher QPS at nearly-zero additional memory overhead (~0.6%). Consequently, Shortcut achieves 1.13× and 1.62× higher memory efficiency than LeanStore and Shortcut-base, respectively.

7 Related Work

SSD-based Tree Index. B+trees [1–4, 16] and LSM-trees [54–56] are the most widely used indexes in modern SSD-based storage systems. Prior works have also introduced on-disk learned indexes [44, 57], but they suffer from similar robustness issues as in-memory learned indexes [20–29]. Thus, several works [51, 52] propose to combine the idea of learned indexes to make indexing faster. Bourbon [51] leverages learned indexes for LSM-trees to accelerate index block lookups in SSTables. However, it still suffers from access dependency across multiple levels, as upper levels hold the freshest data and must be accessed first. In contrast, Shortcut achieves low index latency for B+tree by training ML models to address inherent pointer-chasing dependency.

Solution	Memory (GB)	QPS	QPS/GB
LeanStore	1.60	15244	9528
Shortcut-base	2.76	18216	6600
Shortcut	1.61	17262	10721

Table 1: (Exp #11) Memory efficiency comparison of different solutions. *QPS is SLA-bounded.*

Prefetching techniques. Prefetching is a common technique for reducing access latency and can be categorized into two classes based on their scope [58]. The first class consists of general-purpose prefetchers, which track access history or pattern, mine correlations, and predict future accesses under the prior assumption of temporal and/or spatial locality [59–64]. The second class leverages application-specific knowledge, such as prefetching for linked data structures [17, 65, 66], indirect memory access [67], graph [68, 69], and LSM-trees [70].

For B+tree, existing methods [71–73] are limited to enlarging node size to prefetch multiple blocks within a single level but cannot prefetch across the whole traversal path. For other tree indexes, Cuckoo Trie [74] leverages a hash representation of trie and prefetches the locations of multiple prefix lengths to exploit memory-level parallelism. However, it is a novel and carefully designed in-memory index, whereas Shortcut requires no modification to the underlying SSD-based tree structure. ASAP [75] targets radix tree traversal for page table lookups. It pre-allocates a contiguous virtual memory region for all nodes at the same level, allowing the virtual addresses of all page-table levels to be derived from a given virtual address. This approach enables one-shot traversal for radix trees but is not applicable to other trees.

8 Conclusion

This paper presents Shortcut, an ultra-space-efficient B+tree companion that achieves one-shot traversal by leveraging intra-query parallelism. By proactively predicting the traversal path in advance, Shortcut mitigates the latency bottleneck caused by pointer chasing. It introduces several novel techniques to reduce the excessive space overhead to near zero. Experimental results from both microbenchmarks and production workloads show effective latency reduction without compromising throughput or memory efficiency.

References

[1] Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, pages 107–141, 1970.

[2] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[3] Goetz Graefe. Modern b-tree techniques. *Foundations and Trends® in Databases*, 3(4):203–402, 2011.

[4] Viktor Leis. Leanstore: A high-performance storage engine for nvme ssds. *Proc. VLDB Endow.*, 17(12):4536–4545, August 2024.

[5] Marcus Müller, Lawrence Benson, and Viktor Leis. B-trees are back: Engineering fast and pageable node layouts. *Proc. ACM Manag. Data*, 3(1), February 2025.

[6] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[7] Ohad Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS)*, 3(4):1–27, 2008.

[8] Simran Bijral and Debajyoti Mukhopadhyay. Efficient fuzzy search engine with b-tree search mechanism. *CoRR*, abs/1411.6773, 2014.

[9] Aaron Klotz. Samsung's revived z-nand targets 15x performance increase over traditional nand — once a competitor to intel's optane, z-nand makes a play for ai datacenters | tom's hardware, 8 2025. [Online; accessed 2025-08-11].

[10] Hassam Nasir. Sandisk and sk hynix join forces to standardize high bandwidth flash memory, a nand-based alternative to hbm for ai gpus — move could enable 8-16x higher capacity compared to dram | tom's hardware, 8 2025. [Online; accessed 2025-08-15].

[11] Specifications - nvm express, 1 2020. [Online; accessed 2025-07-15].

[12] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 45–51, 2022.

[13] Gabriel Haas and Viktor Leis. What modern nvme storage can do, and how to exploit it: High-performance i/o for high-performance storage engines. *Proc. VLDB Endow.*, 16(9):2090–2102, May 2023.

[14] Yanbo Zhou, Erci Xu, Anisa Su, Jim Harris, Adam Manzanares, and Steven Swanson. Sleeping with one eye open: Fast, sustainable storage with sandman. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 496–511, New York, NY, USA, 2025. Association for Computing Machinery.

[15] Samsung. PM1743 | Enterprise SSD | Samsung Semiconductor Global. https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1743/, 2025. [Accessed 13-08-2025].

[16] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Chi-Keung Luk and Todd C Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 222–233, 1996.

[18] Li Wang, Zining Zhang, Bingsheng He, and Zhenjie Zhang. Pa-tree: Polled-mode asynchronous B+ tree for nvme. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 553–564. IEEE, 2020.

[19] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. The art of latency hiding in modern database engines. *Proc. VLDB Endow.*, 17(3):577–590, November 2023.

[20] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.

[21] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.*, 13(8):1162–1175, April 2020.

[22] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. Xindex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 308–320, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. Alex: An updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 969–984, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. Finedex: a fine-grained learned index scheme for scalable and concurrent memory systems. *Proc. VLDB Endow.*, 15(2):321–334, October 2021.

[25] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable learned index with precise positions. *Proc. VLDB Endow.*, 14(8):1276–1288, April 2021.

[26] Jiake Ge, Huanchen Zhang, Boyu Shi, Yuanhui Luo, Yunda Guo, Yunpeng Chai, Yuxing Chen, and Anqun Pan. Sali: A scalable adaptive learned index framework based on probability models. *Proc. ACM Manag. Data*, 1(4), December 2023.

[27] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. Dili: A distribution-driven learned index. *Proc. VLDB Endow.*, 16(9):2212–2224, May 2023.

[28] Jin Yang, Heejin Yoon, Gyeongchan Yun, Sam H. Noh, and Young-ri Choi. Dytis: A dynamic dataset targeted index structure simultaneously efficient for search, insert, and scan. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 800–816, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Yuxuan Mo and Yu Hua. Loft: A lock-free and adaptive learned index with high scalability for dynamic workloads. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 475–491, New York, NY, USA, 2025. Association for Computing Machinery.

[30] Yuanhui Luo, Minhui Xie, Yiheng Tong, Shichao Jiang, and Yunpeng Chai. Understanding robustness issues of updatable learned indexes: [experiments & analysis]. *Proc. ACM Manag. Data*, 3(4), September 2025.

[31] Chen Chen, Wenshao Zhong, and Xingbo Wu. Building an efficient key-value store in a flexible address space. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 51–68, New York, NY, USA, 2022. Association for Computing Machinery.

[32] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. Leanstore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 185–196, 2018.

[33] Xiangqun Zhang, Shuyi Pei, Jongmoo Choi, and Bryan S. Kim. Excessive ssd-internal parallelism considered harmful. In Ali Anwar, Ningfang Mi, Vasily Tarasov, and Yiying Zhang, editors, *Proceedings of the 15th ACM/USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2023, Boston, MA, USA, 9 July 2023*, pages 65–72. ACM, 2023.

[34] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage apis: a systematic study of libaio, spdk, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 120–127, 2022.

[35] Jens Axboe. Efficient io with io_uring, 2019.

[36] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.

[37] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.

[38] Chuandong Li, Ran Yi, Zonghao Zhang, Jing Liu, Changwoo Min, Jie Zhang, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Diyu Zhou. Aeolia: A fast and secure userspace interrupt-based storage stack. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, page 479–495, New York, NY, USA, 2025. Association for Computing Machinery.

[39] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. Are updatable learned indexes ready? *Proc. VLDB Endow.*, 15(11):3004–3017, July 2022.

[40] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. Learned index: A comprehensive experimental evaluation. *Proc. VLDB Endow.*, 16(8):1992–2004, April 2023.

[41] Jens Axboe. Flexible I/O Tester, 2022.

[42] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[43] Amazon EC2 I4g instances — aws.amazon.com. https://aws.amazon.com/cn/ec2/instance-types/i4g/. [Accessed 08-12-2025].

[44] Jiaoyi Zhang, Kai Su, and Huanchen Zhang. Making in-memory learned indexes efficient on disk. *Proc. ACM Manag. Data*, 2(3), May 2024.

[45] WiredTiger Storage Engine - Database Manual - MongoDB Docs — mongodb.com. https://www.mongodb.com/docs/manual/core/wiredtiger/. [Accessed 17-09-2025].

[46] Oliver Kowalke. Boost C++ Libraries. https://www.boost.org/doc/libs/1_89_0/libs/coroutine/doc/html/index.html. [Accessed 17-09-2025].

[47] Christian E Lopez and Caleb Gallemore. An augmented multilingual twitter dataset for studying the covid-19 infodemic. *Social Network Analysis and Mining*, 11(1):102, 2021.

[48] Libraries.io - security & maintenance data for open source software — libraries.io. https://libraries.io/. [Accessed 17-09-2025].

[49] Suhas SP Rao, Miriam H Huntley, Neva C Durand, Elena K Stamenova, Ivan D Bochkov, James T Robinson, Adrian L Sanborn, Ido Machol, Arina D Omer, Eric S Lander, et al. A 3d map of the human genome at kilobase resolution reveals principles of chromatin looping. *Cell*, 159(7):1665–1680, 2014.

[50] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Sosd: A benchmark for learned indexes. *NeurIPS Workshop on Machine Learning for Systems*, 2019.

[51] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to bourbon: A learned index for Log-Structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 155–171. USENIX Association, November 2020.

[52] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered Key-Value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 117–135. USENIX Association, November 2020.

[53] TPC-C Homepage — tpc.org. https://www.tpc.org/tpcc/. [Accessed 17-09-2025].

[54] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)*, 17(4):1–32, 2021.

[55] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta informatica*, 33(4):351–385, 1996.

[56] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions On Storage (TOS)*, 13(1):1–28, 2017.

[57] Hai Lan, Zhifeng Bao, J Shane Culpepper, and Renata Borovica-Gajic. Updatable learned indexes meet disk-resident dbms-from evaluations to design choices. *Proceedings of the ACM on Management of Data*, 1(2):1–22, 2023.

[58] Steven P Vanderwiel and David J Lilja. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32(2):174–199, 2000.

[59] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News*, 18(2SI):364–373, 1990.

[60] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 252–263, 1997.

[61] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. Baleen:{ML} admission & prefetching for flash caches. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 347–371, 2024.

[62] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. In *International Conference on Machine Learning*, pages 1919–1928. PMLR, 2018.

[63] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin.

A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 861–873, 2021.

[64] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference-Volume 1*, pages 13–13, 1994.

[65] Amir Roth, Andreas Moshovos, and Gurindar S Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 115–126, 1998.

[66] Magnus Karlsson, Fredrik Dahlgren, and Per Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)*, pages 206–217. IEEE, 2000.

[67] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. Imp: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 178–190, 2015.

[68] Sam Ainsworth and Timothy M Jones. Graph prefetching using data structure knowledge. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–11, 2016.

[69] Karthik Nilakant, Valentin Dalibard, Amitabha Roy, and Eiko Yoneki. Prefedge: Ssd prefetcher for large-scale graph traversal. In *Proceedings of International Conference on Systems and Storage*, pages 1–12, 2014.

[70] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. Leaper: A learned prefetcher for cache invalidation in lsm-tree based storage engines. *Proceedings of the VLDB Endowment*, 13(12):1976–1989, 2020.

[71] Shimin Chen, Phillip B Gibbons, and Todd C Mowry. Improving index performance through prefetching. *ACM SIGMOD Record*, 30(2):235–246, 2001.

[72] Shimin Chen, Phillip B Gibbons, Todd C Mowry, and Gary Valentin. Fractal prefetching b+-trees: Optimizing both cache and disk performance. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 157–168, 2002.

[73] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[74] Adar Zeitak and Adam Morrison. Cuckoo trie: Exploiting memory-level parallelism for efficient dram indexing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 147–162, 2021.

[75] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched address translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1023–1036, 2019.